LibWiiEsp: Manual de instalación y uso

Ezequiel Vázquez de la Calle

Versión 0.9.2

Licencia

Índice

Índice de figuras

Índice de cuadros

Resumen

Debido a las diferencias existentes entre el desarrollo para un PC ordinario y una videoconsola, en este caso, para Nintendo Wii, se hace patente la necesidad de recoger todos esos detalles que, siendo sencillos de solventar, pueden suponer más de un quebradero de cabeza a un programador sin ningún tipo de experiencia en la programación para Wii. Además, la propia construcción de *LibWiiEsp* implica una serie de consideraciones a tener en cuenta a la hora de sacarle el máximo rendimiento. El objetivo de este documento es recoger todos los pormenores que un programador debe tener en cuenta a la hora de crear un videojuego para la consola, utilizando *LibWiiEsp* como herramienta.

1. Instalación del entorno de desarrollo

Para comenzar el desarrollo de un videojuego para Wii utilizando como herramienta *LibWiiEsp*, es necesario instalar en el sistema todas las dependencias que la biblioteca necesita.

Esta sección del manual detalla paso a paso la instalación de estas dependencias en un sistema *GNU/-Linux* de 32 bits, siendo el proceso prácticamente el mismo para sistemas de 64 bits (únicamente cambia la versión de las herramientas base, que deberá ser en este caso la adecuada para 64 bits). Para sistemas Windows también es posible desarrollar con *LibWiiEsp*, pero este manual no cubre este tipo de arquitecturas.

En primer lugar, hay que crear una carpeta accesible para todos los usuarios del sistema, donde irán emplazadas las herramientas y dependencias de *LibWiiEsp*. Lo ideal es crear un directorio dentro de `/opt` y otorgarle a éste los permisos necesarios, pero es posible realizar la instalación en nuestro directorio `/home`, de tal manera que sólo nuestro usuario pueda acceder a estas herramientas.

Suponiendo que se escoge la primera opción, se crea el directorio en `/opt` y se le asignan permisos para que todos los usuarios del sistema puedan hacer uso de lo que en él se almacene:

```
sudo mkdir /opt/devkitpro
sudo chmod 777 /opt/devkitpro
```

A continuación, hay que descargar las herramientas que sirven de base para *LibWiiEsp*, que son el conjunto de compiladores, bibliotecas y binarios *DevKitPPC*, y la biblioteca de bajo nivel *libOgc*, en su versión 1.8.4, junto con la modificación de *libFat* 1.0.7 compatible con ella. Ambos recursos se encuentran disponibles en la forja del proyecto, en el paquete *Dependencias*. Se deben descargar en el directorio que acabamos de crear:

```
cd /opt/devkitpro
wget http://forja.rediris.es/frs/download.php/2316/devkitPPC_r21-i686-linux.tar.bz2
wget http://forja.rediris.es/frs/download.php/2315/libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

El siguiente paso es descomprimir ambos ficheros y, si no queremos tener ocupado espacio innecesariamente, eliminarlos. Las instrucciones para ejecutar estas acciones son:

```
tar -xvjf devkitPPC_r21-i686-linux.tar.bz2
tar -xvzf libogc-1.8.4-and-libfat-1.0.7.tar.gz
rm devkitPPC_r21-i686-linux.tar.bz2 libogc-1.8.4-and-libfat-1.0.7.tar.gz
```

Después de esto, es necesario establecer algunas variables de entorno para que el sistema sepa dónde localizar estas herramientas que acabamos de instalar. Las dos primeras consisten en la ruta hasta el

directorio base *devkitpro*, y hasta el directorio donde se encuentran las herramientas como tal, concretamente *devkitpro/devkitPPC*. La tercera variable de entorno indica la dirección IP de nuestra Nintendo Wii dentro de la red local, dato necesario para ejecutar correctamente las aplicaciones que se desarrollen sin que haga falta copiar el ejecutable en la tarjeta SD de la consola cada vez que se genere uno nuevo.

Para establecer estas variables de entorno, basta con editar el archivo de configuración */.bashrc* del usuario con el que vayamos a desarrollar utilizando *LibWiiEsp*. Si por alguna causa fuera necesario que todos los usuarios del sistema tengan que usar estas herramientas, el archivo de configuración a editar sería */etc/bashrc*, o su equivalente en el sistema (por ejemplo, para una distribución *Ubuntu 10.10* de 32 bits, el archivo es */etc/bash.bashrc*).

Suponiendo que sólo nuestro usuario va a desarrollar utilizando *LibWiiEsp*, las instrucciones para establecer estas variables de entorno serían:

```
echo export DEVKITPRO=/opt/devkitpro >> ~/.bashrc
echo export DEVKITPPC=$DEVKITPRO/devkitPPC >> ~/.bashrc
echo export WIILOAD=tcp:192.168.X.X >> ~/.bashrc
```

Un detalle, en estas instrucciones, *192.168.X.X* se debe sustituir por la dirección IP que tiene asignada la consola Nintendo Wii en la red local.

A continuación, podemos recargar el fichero de configuración */.bashrc* para tener listas las varibles de entorno mediante la orden:

```
source ~/.bashrc
```

Llegados a este punto, el último paso para que el entorno de desarrollo funcione correctamente con *LibWiiEsp* es instalar la biblioteca propiamente dicha. Existen dos maneras de realizar esto, o bien podemos descargar la versión estable publicada en la forja, o también compilando el código disponible desde ésta. Para el primer caso, basta con descargar el paquete comprimido desde la página principal de la Forja del proyecto (http://forja.rediris.es/projects/libwiiesp), copiarlo al directorio */opt/devkitpro*, y descomprimirlo allí.

Para compilar la biblioteca a partir de los fuentes, debemos ejecutar las siguientes líneas, siempre que se hayan instalado previamente las dependencias:

```
svn co https://forja.rediris.es/svn/libwiiesp/trunk libwiiesp
cd libwiiesp
make install
```

Y después de la instalación de la biblioteca, ya tenemos preparado nuestro sistema para comenzar con el desarrollo de videojuegos en dos dimensiones para Nintendo Wii.

2. Estructura de un proyecto con *LibWiiEsp*

Una vez instalado correctamente el entorno de desarrollo para utilizar *LibWiiEsp*, el siguiente paso es crear la estructura de archivos y directorios necesarios para trabajar con un nuevo proyecto para Nintendo Wii. Por supuesto, queda en manos del programador la decisión final sobre esta estructura de directorios, pero, debido a todo lo que hay que tener en cuenta a la hora de desarrollar con *LibWiiEsp* (sobretodo, en lo referente a la compilación y enlazado de los archivos que componen un proyecto), voy a presentar en esta sección un ejemplo de proyecto vacío que quedaría listo para comenzar el desarrollo. Además de tratar la estructura de directorios, también mostraré y explicaré el código de un archivo *makefile* que pueda trabajar con esta organizción para el proyecto.

2.1. Estructura de directorios

En primer lugar, hay que crear un directorio base donde almacenar todo lo relativo a nuestro proyecto. La localización de este directorio en el sistema es indiferente, de tal manera que, para el ejemplo, lo haremos en nuestro directorio */home* con las instrucciones:

```
mkdir ~/juego
cd ~/juego
```

Una vez dentro, la siguiente estructura de directorios sería más que suficiente para albergar todos los componentes del proyecto:

- *doc*: Documentación del proyecto.
- *lib*: Bibliotecas externas que se vayan a utilizar en el proyecto. Aquí se debe guardar cada biblioteca en un directorio separado. En cada directorio debe existir un archivo *makefile* el cual compile la biblioteca, y genere un archivo de enlazado estático *.a*, además los archivos de cabecera de la biblioteca externa tienen que estar justo bajo este directorio principal de la biblioteca externa.
- *media*: En este directorio se colocarán todos los archivos que contengan recursos multimedia que vayan a ser empleados en el proyecto. De momento, sólo están soportados imágenes *bitmap* de 24 bits de color directo, fuentes de texto soportadas por *FreeType2*, efectos de sonido en formato *PCM*, y pistas de música *mp3*.
- *src*: Aquí van los archivos fuente del proyecto.
- *xml*: Archivos de datos del proyecto. Como mínimo, aquí se encontrarán el archivo que describe los recursos multimedia que se cargarán en la galería de medias, el archivo del soporte de idiomas, y el archivo de configuración del programa.

2.2. El archivo *makefile*

La estructura de directorios, y los detalles que la acompañan (como las restricciones de estructuración a la hora de utilizar bibliotecas externas en el proyecto), se han definido así para trabajar con un archivo de compilación *makefile* concreto.

Este archivo de compilación implica una serie de modificaciones considerables en comparación con uno que genere un ejecutable para PC en entornos *GNU/Linux*, por lo que va a detallarse su funcionamiento sección a sección. El objetivo de este *makefile* es generar un ejecutable con extensión *.dol*, que puede lanzarse en una videoconsola Nintendo Wii. A continuación, se muestra un sencillo ejemplo de *makefile* compatible con la estructura de directorios planteada en el punto anterior:

```
# Informacion configurable
LOCALLIBS = tinyxml bullet
PROJECT = Wii Pang
TARGET = boot
BUILD = build
SOURCE = src
DEPSDIR = $(BUILD)

# Reglas de compilacion
.SUFFIXES:
include $(DEVKITPPC)/wii_rules
include $(DEVKITPPC)/libwiiesp_rules

# Generar una lista con todos los ficheros objeto del proyecto
CPPFILES = $(notdir $(wildcard $(SOURCE)/*.cpp))
OFILES = $(CPPFILES:.cpp=.o)
OBJS = $(addprefix $(CURDIR)/$(BUILD)/,$(OFILES))

# Variables para la compilacion
INCLUDE = $(foreach dir,$(LOCALLIBS),-Ilib/$(dir)) -I$(LIBOGC_INC)
LIBPATHS = -L$(LIBOGC_LIB) -L$(CURDIR)/$(BUILD)
VPATH = $(SOURCE)$(foreach dir,$(LOCALLIBS),:lib/$(dir))
LIBS = $(LIBWIIESP_LIBS) -ltinyxml -lbullet
OUTPUT = $(CURDIR)/$(TARGET)
LD = $(CXX)

# Flags para la compilacion
CXXFLAGS        = -g -ansi -Wall $(MACHDEP) $(INCLUDE) -std=c++0x
OPTIONS         = -MMD -MP -MF
LDFLAGS         = -g $(MACHDEP) -Wl,-Map,$(notdir $@).map

# Objetivos
.PHONY: all $(BUILD) libs $(LOCALLIBS) listo run clean

all: $(BUILD) libs $(OUTPUT).dol listo

$(BUILD):
	@[ -d $(BUILD) ] || mkdir -p $(BUILD)

libs: $(LOCALLIBS)
	@echo Bibliotecas externas listas

$(LOCALLIBS):
	$(MAKE) --no-print-directory --silent -C lib/$@
	mv lib/$@/*.a $(BUILD)

$(CURDIR)/$(BUILD)/%.o: $(CURDIR)/$(SOURCE)/%.cpp
	$(CXX) -MMD -MP -MF $(DEPSDIR)/$*.d $(CXXFLAGS) -c $< -o $@

listo:
	$(RM) $(SOURCE)/-g $(OUTPUT).elf.map

run:
	wiiload $(TARGET).dol

```

```
clean:
	for dir in $(LOCALLIBS); do $(MAKE) clean -C lib/$$dir; done
	$(RM) -fr $(BUILD) $(OUTPUT).elf $(OUTPUT).dol *~ $(SOURCE)/*~

DEPENDS :=	$(OBJS:.o=.d)

$(OUTPUT).dol: $(OUTPUT).elf

$(OUTPUT).elf: $(OBJS)

-include $(DEPENDS)
```

En el primer bloque que aparece en el ejemplo, se definen una serie de variables que indican los directorios que forman parte del proyecto. *LOCALLIBS* debe recoger los nombres, separados por un espacio, de los directorios principales de cada biblioteca externa que se vaya a emplear en la compilación. La variable *PROJECT* indica el nombre completo del proyecto, y *TARGET* el nombre, sin extensión, del ejecutable que se generará. Para que el programa pueda ser ejecutado con *HomeBrew Channel*, se recomienda que se nombre *boot*. Por último, *BUILD* indica el directorio que se creará cuando se ordene la compilación del proyecto para almacenar todos los ficheros objeto del proyecto, *SOURCE* es el directorio donde están todos los archivos fuente, y *DEPSDIR* es donde se generarán los archivos que indican las dependencias entre módulos del sistema, que debe ser el mismo directorio que *BUILD*.

A continuación, se deben importar las reglas de compilación para Nintendo Wii, tanto las necesarias para utilizar las herramientas de *DevKitPPC*, como las propias de *LibWiiEsp*. Para ello, antes hay que limpiar las reglas implícitas existentes, lo cual se consigue con la instrucción *.SUFFIXES:*.

El bloque de tres instrucciones que viene a continuación se encarga de almacenar en variables una lista de todos los archivos *.cpp* que se encuentran en el directorio de fuentes, y otra lista con la ruta absoluta de los ficheros objeto que se generarán al compilar (es decir, archivos con extensión *.o* en la carpeta indicada por la variable *BUILD*).

El cuarto bloque de instrucciones recoge los directorios donde se encuentran los archivos de cabeceras externas (los almacena en la variable *INCLUDE*), los directorios en los cuales hay que buscar las bibliotecas de enlazado estático (guardados en la variable *LIBPATHS*), establece el *VPATH*, crea la lista de bibliotecas a enlazar estáticamente (tomando la variable *LIBWIIESP_LIBS* para contar con todo lo que necesita un ejecutable generado con *LibWiiEsp*, pero se le debe añadir además lo necesario para enlazar también las bibliotecas externas), y por último, indica la ruta absoluta hasta el ejecutable sin extensión y define el enlazador que se utilizará.

El bloque siguiente establece los flags de compilación necesarios para generar un ejecutable de extensión *.elf*, que posteriormente se transformará a otro con extensión *.dol*. Y justo después comienzan los objetivos del *makefile*, que se describen a continuación:

- *all*: Crea un ejecutable *.dol* a partir del código fuente que se encuentre en el directorio indicado en la variable *SOURCE*. Es el objetivo por defecto al ejecutar *make*.
- *$(BUILD)*: Crea el directorio donde se crearán todos los ficheros objetos del proyecto.
- *libs* y *$(LOCALLIBS)*: Se encargan de compilar las bibliotecas externas, de empaquetarlas en ficheros de enlace estático y mover éstos al directorio de los ficheros objeto.

- *$(CURDIR)/$(BUILD)/%.o*: Genera un fichero objeto en el directorio *$(BUILD)* por cada módulo que se encuentre en el directorio de los ficheros fuentes.
- *listo*: Objetivo que limpia un archivo de extensión *.elf.map*.
- *run*: Lanza la utilidad *wiiload* para enviar el ejecutable a la Nintendo Wii. Ésta debe tener abierto el *HomeBrew Channel* y estar conectada a la red local con la misma dirección IP que se indicó en la variable de entorno *WIILOAD*, de otro modo no ejecutará el programa.
- *clean*: Objetivo que limpia de archivos temporales el proyecto. Elimina la carpeta donde se almacenan los ficheros objeto, y ejecuta también los objetivos *clean* de cada biblioteca externa.

La última parte de este ejemplo se encarga de comprobar que se cumplan las dependencias de los módulos a compilar, de generar un ejecutable *.elf* a partir de los módulos objeto del proyecto, y en último lugar, de crear el ejecutable definitivo *.dol* a partir del archivo *.elf*.

2.3. Ejecución del programa

Una vez generado nuestro programa, existen dos maneras de ejecutarlo en la videoconsola. La primera, ya descrita, es lanzarlo a través del objetivo *make run*, para lo cual necesitamos tener correctamente conectada la Nintendo Wii a la red local, y *wiiload* bien configurado con la IP de la consola.

La otra manera de hacerlo, que es la necesaria para poder distribuir el programa, es guardar el ejecutable en un directorio de la tarjeta SD de la consola, cuya ruta debe ser */apps/XXX/boot.dol*, donde XXX es el nombre unix de nuestra aplicación (importante que no contenga espacios). Nótese que tanto el nombre *boot.dol* como el hecho de que el directorio que lo contiene esté dentro del directorio */apps* es obligatorio.

Debido a las características de *HomeBrew Channel*, podemos acompañar nuestro ejecutable con una imagen que servirá de icono para la aplicación (debe tener formato PNG, llamarse *icon.png*, y tener un tamaño de 128 píxeles de ancho y 48 de alto), y un fichero XML con la información que deseemos aportar sobre el programa (debe ser un XML con un formato concreto, llamado *meta.xml*). Ambos ficheros, la imagen y el archivo de datos, deben ir en el mismo directorio que el ejecutable.

Una referencia sobre los nodos del fichero de datos *meta.xml* que acompaña a una aplicación se puede encontrar a continuación:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<app version="1">
  <name>El nombre de la aplicacion</name>
  <coder>Autor o autores del codigo de la aplicacion.</coder>
  <version>La version de la aplicacion.</version>
  <release_date>Dia de lanzamiento. Formato: AAAAMMDD</release_date>
  <short_description>Comentario que se muestra en el menu principal. No se
     recomienda que sea mayor de 30 caracteres</short_description>
  <long_description>Descripcion detallada del programa</long_description>
</app>
```

3. Consideraciones sobre programar con *LibWiiEsp*

A la hora de programar para una plataforma cerrada como es Nintendo Wii, hay que tener una serie de aspectos en cuenta, debido a la arquitectura concreta que tiene el hardware de la consola. Es muy importante tener en cuenta estos detalles, ya que en caso contrario se pueden producir errores o comportamientos inesperados, como texturas que se pisan entre sí en la memoria principal, o excepciones propias del sistema.

A continuación se describen todos esos pequeños detalles a tener en cuenta, que si bien sólo suponen un cambio en algunos hábitos a la hora de programar, nos aseguran que todo irá bien (al menos, en lo que al funcionamiento del hardware se refiere).

3.1. Big Endian

Lo primero que hay que tener en cuenta es que la representación de los datos en la consola Nintendo Wii se realiza con Big Endian, al contrario que las plataformas Intel (la arquitectura, no la marca del procesador), que utilizan Little Endian para almacenar la información. Cuando un dato ocupa más de un byte, se puede organizar de mayor a menor peso (esto sería Big Endian), o bien de menor a mayor peso (que es la organización en Little Endian).

Un ejemplo muy claro es la representación del *número mágico* que emplean los archivos de imágenes formadas por mapas de bits (también conocidos como *bitmaps*). Este número es, en su representación decimal, 19778. Si esta cifra la representamos en sistema hexadecimal con Little Endian el resultado sería 0x4D42; sin embargo, en un sistema que utilice Big Endian, esta cifra se representaría como el hexadecimal 0x424D en la memoria principal.

Así pues, como consecuencia de esto, siempre que queramos cargar un archivo binario (una imagen, una pista de música, etc.) en la consola Nintendo Wii, tenemos dos opciones: o bien modificamos la representación del archivo en la plataforma de origen (que normalmente será un ordenador con arquitectura Intel) para que el recurso esté ya representado como Big Endian, o podemos utilizar las funciones del espacio de nombres *endian* que se proporcionan en el archivo de cabecera *util.h* de *LibWiiEsp*. Este espacio de nombre aporta las siguientes dos funciones, que se encargan de transformar variables de 16 o 32 bits entre Little Endian y Big Endian (soportan ambas transformaciones):

```
u16 inline swap16(u16 a)
{
  return ((a<<8) | (a>>8));
}

u32 inline swap32(u32 a)
{
  return ((a)<<24|(((a)<<8) & 0x00FF0000)|(((a)>>8) & 0x0000FF00)|(a)>>24);
}
```

Así pues, únicamente utilizando estas dos funciones con cada dos o cuatro bytes cuando queramos cargar un recurso desde la tarjeta SD podremos evitar los problemas derivados de los distintos sistemas de representación. Para un ejemplo práctico sobre el uso de estas dos funciones, ver el código fuente de la clase Imagen de *LibWiiEsp*.

3.2. Los tipos de datos

A la hora de programar para Nintendo Wii se utilizan siempre estos tipos de datos:

Tipo de dato	Descripción	Rango
u8	Entero de 8 bits sin signo	0 a 255
s8	Entero de 8 bits con signo	-127 a 128
u16	Entero de 16 bits sin signo	rango 0 a 65535
s16	Entero de 16 bits con signo	-32768 a 32767
u32	Entero de 32 bits sin signo	0 a 0xffffffff
s32	Entero de 32 bits con signo	-0x80000000 a 0x7fffffff
u64	Entero de 64 bits sin signo	0 a 0xffffffffffffffff
s64	Entero de 64 bits con signo	-0x8000000000000000 a 0x7fffffffffffffff
f32	Flotante de 32 bits	-
f64	Flotante de 64 bits	-

Cuadro 1: Tipos de datos que se deben emplear al programar para Wii

Todos los tipos de datos disponibles se encuentran en */opt/devkitpro/libogc/include/gctypes.h*.

3.3. La alineación de los datos

Otro detalle importante que se debe contemplar a la hora de programar para Nintendo Wii es que el procesador de la consola necesita alinear los datos conforme a su tamaño. Es decir, si vamos a trabajar con un entero de 32 bits (4 bytes), sólo se pueden almacenar en posiciones de memoria múltiplos de cuatro: 0, 4, 8, 12... Lo mismo ocurre con variables de 16 o 64 bits.

Para ilustrar el comportamiento del procesador respecto a la alineación de los datos, podemos considerar el siguiente ejemplo:

```
struct ejemplo {
  u8 entero;
  u32 otro;
}
```

La estructura *ejemplo* no se representará igual compilando en una plataforma Intel que en la Wii, ya que en la primera, *entero* se situará en la posición 0 de la memoria, y *otro* ocupará de la posición 1 hasta la 5. Sin embargo, en nuestra consola *entero* ocupará el mismo lugar, la posición 0, pero la variable *otro* necesitará estar alineada a su tamaño, es decir, ocupará las posiciones 4 a 7 de la memoria, y las posiciones entre *entero* y *otro* se rellenarán con ceros para asegurar que el entero de 32 bits está alineado.

Esto revierte tanto en la cantidad de memoria ocupada, como en el hecho de que si el procesador encontrara un dato desalineado e intentara leerlo, se produciría un error en el sistema.

Para solventar este inconveniente, se debe jugar inteligentemente con las declaraciones de las variables, de tal manera que se organicen los datos de forma alineada.

3.4. Alineación de datos que usan DMA

Este es otro caso en el que influye la necesidad de alinear los datos con los que se trabaja. El procesador de la Nintendo Wii trabaja con datos cacheados, pero no así los periféricos (la tarjeta SD, los dispositivos USB o el lector DVD). Además, estos periféricos trabajan con una alineación fija de 32 bytes, y si no se contempla este detalle, se pueden producir errores de machacamiento de datos al realizar dos lecturas consecutivas desde periféricos.

Además, cuando leemos un dato desde un periférico, se corre el riesgo de machacar el contenido de la caché del procesador, por lo que es imprescindible volcar explícitamente los datos leídos en la memoria, es decir, asegurar de que la lectura se ha realizado completamente antes de realizar cualquier otra acción.

Es muy sencillo evitar esta situación, y es preparando la memoria en la que se almacenarán los datos leídos desde el periférico, de tal manera que esté alineada a 32 bytes y su tamaño sea múltiplo de 32. Para ilustrar cómo hacer esto, se muestra un ejemplo a continuación:

```
// Abrimos el archivo mediante un flujo
ifstream archivo;
archivo.open("SD:/apps/wiipang/media/sonido.pcm", ios::binary);

// Obtener el tamano del sonido
archivo.seekg(0, ios::end);
u16 size = archivo.tellg();
archivo.seekg(0, ios::beg);

// Calcular el relleno que hay que aplicar a la memoria reservada
// para que su tamano sea multiplo de 32
u8 relleno = (size * sizeof(s16)) % 32;

// Reservar memoria alineada: utilizamos memalign en lugar de malloc
s16* sonido = (s16*)memalign(32, size * sizeof(s16) + relleno);

// Realizar la lectura de datos desde el periferico
// Utilizar char* como tipo de lectura es por compatibilidad con libFat
archivo.read((char*)sonido, size);

// Fijar los datos leidos en la memoria, evita machacamiento en la cache
DCFlushRange(sonido, size * sizeof(s16) + relleno);
```

Como puede verse en el ejemplo, se calcula en primer lugar el relleno necesario para que la memoria que ocupa el archivo binario a cargar sea múltiplo de 32 bytes, a continuación se reserva la memoria alineada a 32 bytes utilizando la función *memalign* en lugar de *malloc*, y el último paso, tras leer la información desde el archivo, consiste en realizar el volcado explícito de información desde la caché de lectura a la memoria, utilizando la función *DCFlushRange* (esta función reibe la dirección de memoria a la que se quiere realizar el volcado, y el tamaño de ésta).

3.5. Depuración con *LibWiiEsp*

Con toda la información anterior descrita en este manual y la referencia completa de la biblioteca, se puede comenzar a desarrollar un videojuego sencillo. Como en todo proceso de desarrollo de software, ocurrirán errores, y aquí es donde se hace patente la falta de medios disponibles para depurar el código.

Existen dos tipos de errores que podremos encontrarnos a la hora de programar para Nintendo Wii, que corresponden con las fases de compilación y ejecución. A continuación, vamos a plantear cómo solucionar los posibles errores que pueden surgir en cada uno de estos momentos:

3.5.1. Errores de compilación

En la fase de compilación suelen ocurrir, sobretodo, errores de sintaxis, aunque también es posible que ocurran errores de enlazado si las rutas hasta los archivos de cabeceras o las bibliotecas de enlace estático no son correctas. El compilador nos avisará de cualquier tipo de error que ocurra, tanto en el preprocesado, como en la compilación propiamente dicha y en el enlazado. Así pues, prestando atención a los mensajes que pueda proporcionarnos el compilador sobre los errores o avisos que se den, podremos depurar el código fuente de nuestra aplicación.

Sobre los mensajes de enlazado, el enlazador nos proporcionará información suficiente para saber qué ha fallado durante esta operación, pero si se sigue al pie de la letra este manual (especialmente, la instalación del entorno y la creación del *makefile*) no debería haber ningún problema.

3.5.2. Errores de ejecución

En la fase de ejecución es cuando tenemos verdaderos problemas para depurar nuestra aplicación, y es que apenas hay herramientas que puedan facilitarnos el conocer el estado de los objetos del sistema, el contenido de las variables, etc.

LibWiiEsp proporciona un sistema de registro de eventos del sistema, la clase *Logger*, que permite que escribamos un log de errores, avisos e información variada. La forma de trabajar con esta clase es muy sencilla, y todos los detalles pueden consultarse en su documentación (ver referencia completa de la biblioteca). Pero hay ocasiones en que los errores en tiempo de ejecución requieren más precisión que una serie de mensajes que aportemos desde nuestro propio código, ya que el uso de la clase *Logger* está recomendado en casos de comportamiento inesperado del programa, pero no de errores como tal.

Figura 1: Ejemplo de pantalla de error en tiempo de ejecución

Cuando ocurre un error de ejecución en la Nintendo Wii, ésta nos presentará una pantalla de error parecida a la imagen de la figura 1. En esta pantalla de error pueden apreciarse tres partes principales. A la hora de localizar en qué parte de nuestro código se ha provocado este error, necesitamos fijarnos en la sección *STACK DUMP*, es decir, la segunda. Esta parte del mensaje de error nos detalla la traza de llamadas a función que se han realizado hasta llegar a la llamada que ha producido el error, estando cada elemento de la traza representado por una dirección de memoria en hexadecimal.

Esta informacion es muy útil, ya que podemos localizar a qué línea de nuestro código fuente corresponde cada llamada con una utilidad incluida en *DevKitPPC*, y esta es la utilidad *powerpc-eabi-addr2line*. Se puede localizar en la carpeta */opt/devkitpro/devkitPPC/bin* si se ha seguido este manual para instalar *LibWiiEsp*. Lo más cómodo es crear un enlace simbólico a esta utilidad en el directorio principal de nuestro proyecto, aunque también se puede añadir el directorio *devkitPPC/bin* a la ruta donde el sistema busca ejecutables.

Siguiendo el ejemplo de la figura 1, si queremos saber a qué archivo y qué línea de código pertenece la dirección *0x80011f54*, basta con ejecutar la siguiente línea de código en el directorio principal del proyecto:

```
./powerpc-eabi-addr2line -e boot.elf -f 0x80011f54
```

Para que esta utilidad funcione, debemos tener en el directorio donde la ejecutamos una copia del ejecutable *.elf* generado por nuestro proyecto (el mismo que hemos ejecutado en la consola y que nos ha dado el mensaje de error de la Figura 1). El parámetro *-e* recibe el nombre de este ejecutable *.elf*, y el parámetro *-f* es la dirección de memoria (en hexadecimal) que ha producido el error.

Un ejemplo del resultado de la ejecución de la utilidad *addr2line* puede ser el siguiente:

```
Actor
/home/rabbit/Escritorio/libwiiesp/src/actor.cpp:27
```

Lo cual nos indica que el error se ha producido en la línea 27 del fichero fuente actor.cpp de *LibWiiEsp*.

Un detalle a comentar es que, para salir de la pantalla de error que se muestra en la Figura 1, basta con pulsar el botón *Reset* de la consola, lo que nos devolverá al *HomeBrew Channel*.

En resumen, con la herramienta *addr2line* y la clase *Logger* podemos ir realizando una decente depuración de nuestra aplicación, que aunque hay que reconocer que no es muy cómodo, es mejor que ir dando palos de ciego.

4. Plantillas de *LibWiiEsp*

Con todo lo visto hasta el momento en este manual es posible comenzar el desarrollo de nuestra aplicación, ya que tenemos las herramientas preparadas, sabemos qué hay que tener en cuenta a la hora de utilizarlas, y además contamos con las clases de *LibWiiEsp*, que nos facilitan (mucho) la vida a la hora de cargar recursos multimedia, establecer el soporte de idiomas, acceder a la tarjeta SD de la consola, dibujar texturas y animaciones en pantalla, etc.

Pero el punto más interesante de *LibWiiEsp*, además de la abstracción que nos proporciona a la hora de trabajar con estos subsistemas de la videoconsola, está formado por las tres plantillas (clases abstractas) que ofrece para facilitar el desarrollo de un videojuego. Estas plantillas, que son *Actor*, Nivel y *Juego*, permiten crear los distintos tipos de actores y niveles, además de la clase principal de nuestro programa, de una manera sencilla y efectiva.

Cabe destacar que, en el caso de que las plantillas ofrecidas no se adaptaran a las necesidades del videojuego que tenemos en mente, siempre podemos personalizar la clase que corresponda al derivarla, o bien modificando la clase original desde el código fuente de la biblioteca.

En esta sección del manual vamos a desgranar los detalles necesarios para poder sacar el máximo jugo a estas tres plantillas que nos facilitarán el desarrollo de nuestro videojuego:

4.1. Sistemas de coordenadas

En primer lugar, hay que comprender cómo funcionan los distintos sistemas de coordenadas que nos encontraremos a la hora de crear el universo de nuestro videojuego. La siguiente imagen ilustra los tres sistemas que pueden darse a la vez en el juego:

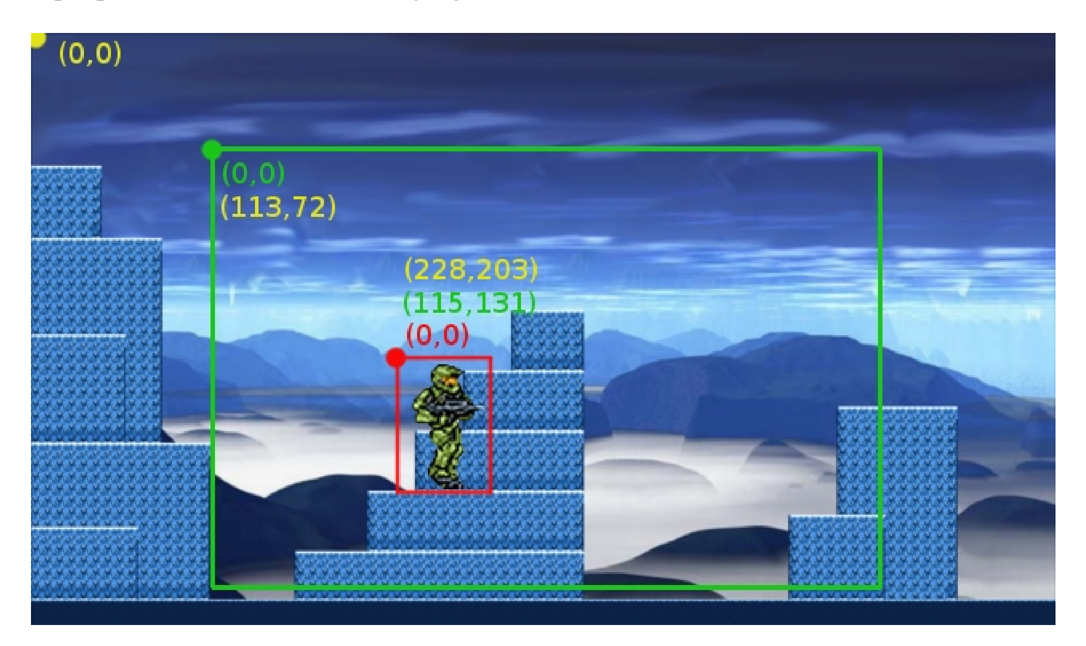


Figura 2: Distintos sistemas de coordenadas en el universo del juego

La imagen al completo representa un escenario completo de un juego. Este escenario es el universo del juego en sí, tiene forma rectangular, y su esquina superior izquierda es el origen de coordenadas, siendo la X positiva hacia la derecha, y la Y hacia abajo. Todos las parejas de coordenadas, en cualquiera

de los tres sistemas que se emplean, se refieren al punto superior izquierdo del objeto al que pertenecen las coordenadas. En la imagen, se distinguen las coordenadas relativas al escenario con el color amarillo.

El rectángulo verde que se aprecia en el centro de la imagen representa la parte del escenario que se muestra en la pantalla (actua como si fuera una ventana desplazable). La pareja de coordenadas (x,y) en amarillo de este rectángulo indica la posición de desplazamiento de la pantalla, o *scroll*, respecto al punto (0,0) del escenario. Jugando con el *scroll* conseguimos que la pantalla *se mueva* sobre el escenario, y muestre la parte que queramos de éste. Por otro lado, el vértice superior izquierdo de la pantalla es el origen de otro sistema de coordenadas, que indica la posición de un objeto (un actor, por ejemplo) dentro de la visualización en pantalla.

Teniendo en cuenta estos dos sistemas de coordenadas, se entiende fácilmente que el actor (el rectangulo rojo) tenga una pareja de coordenadas que indiquen su posición en el escenario (coordenadas en amarillo), y otra pareja de coordenadas (en verde) que denotan su posición en la pantalla. Sin embargo, las coordenadas de un actor respecto a la pantalla no se almacenan, si no que se calculan restando su posición en el escenario (amarillo) menos la posición del desplazamiento de la pantalla (coordenadas amarillas de la pantalla).

Además, el vértice superior izquierdo del rectángulo que representa al actor es el origen de un tercer sistema de coordenadas, que se utiliza como referencia para las cajas de colisión asociadas al actor.

4.2. Actores

Un actor es un objeto que tiene entidad propia dentro del universo del videojuego. En este sentido, son actores tanto los protagonistas manejados por los jugadores, como los enemigos controlados por la máquina, los items que recogemos, cada una de las balas (en el caso de un juego de disparos) también es un actor…

Entrando en el apartado técnico, un actor se representa como un objeto que tiene una pareja de coordenadas (x,y) respecto al origen del escenario, una velocidad en píxeles por fotograma (tanto vertical como horizontal), un conjunto de estados, cada uno de los cuales tiene asociada una animación y varias figuras de colisión, y un indicador sobre qué estado de los posibles es el actual.

A continuación se explican los diversos aspectos a tener en cuenta a la hora de crear un actor utilizando la clase abstracta que proporciona *LibWiiEsp*. En primer lugar, tenemos que crear una clase derivada de *Actor*. En el constructor de nuestra clase derivada, no debemos olvidar pasarle al constructor de *Actor* la cadena de caracteres con la ruta absoluta hasta el archivo de datos desde el que se cargan los datos del actor, y una referencia al nivel en el que participará. Además, tenemos que definir el método *actualizar()*, que es un método virtual puro, y en el cual tenemos que definir el comportamiento del actor dependiendo de su estado actual.

Ambos aspectos se explican con detalle en los siguientes apartados:

4.2.1. Cargando los datos de un actor

Cada actor que se cree derivando la clase *Actor* cargará toda la información relativa a él a través del método *cargarDatosIniciales()*, definido en la propia clase base *Actor* (al cual se llama desde el constructor de ésta clase), y que recibe la ruta absoluta, en la tarjeta SD, de un archivo XML con un formato como el siguiente:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<actor vx="3" vy="3" tipo="jugador">
  <animaciones>
    <animacion estado="normal" img="chief" sec="0" filas="1" columnas="6"
        retardo="3" />
    <animacion estado="mover" img="chief" sec="0,1,2,3,4" filas="1"
        columnas="6" retardo="3" />
    <animacion estado="muerte" img="chief" sec="5" filas="1" columnas="6"
        retardo="0" />
  </animaciones>
  <colisiones>
    <rectangulo estado="normal" x1="27" y1="21" x2="55" y2="21" x3="55" y3=
        "96" x4="27" y4="96" />
    <circulo estado="normal" cx="41" cy="13" radio="8" />
    <rectangulo estado="mover" x1="27" y1="21" x2="55" y2="21" x3="55" y3="
        96" x4="27" y4="96" />
    <circulo estado="mover" cx="41" cy="13" radio="8" />
    <sinfigura estado="muerte" />
  </colisiones>
</actor>
```

En el archivo XML anterior pueden observarse dos grandes bloques, uno para las animaciones y otro para las figuras de colisión. A cada estado (en el ejemplo hay tres, *normal*, *mover* y *muerte*) le corresponde una única animación, pero puede tener ninguna, una o varias figuras de colisión. En el caso de que un estado concreto no tenga ninguna figura de colisión asociada, basta con introducir un nodo con el nombre *sinfigura* y su correspondiente atributo *estado*. Para más información sobre las animaciones o las figuras de colisión, consultar las secciones correspondientes en la referencia de la biblioteca.

Muy importante: Los estados en los cuales puede encontrarse un actor vienen definidos por los que aparezcan en este archivo de datos, y es **imprescindible** que definamos el estado *normal* (al menos, en las animaciones), ya que es el estado que un actor toma por defecto, y si no se encontrara entre los datos del actor, se produciría un error en el sistema.

Para más información sobre la carga de datos iniciales, consultar la documentación de la clase Actor.

4.2.2. Definir el comportamiento de un actor

El comportamiento de un actor, como ya se ha comentado, depende del estado en el que se encuentre. A la hora de crear una clase derivada de *Actor* se deberían implementar tantos métodos como estados se hayan definido en el archivo de datos del actor, de tal manera que cada uno de estos métodos corresponda con el comportamiento esperado en cada uno de los estados.

Por ejemplo, si tenemos un actor con tres estados (normal, caer y mover), tendríamos tres nuevos métodos llamados *estado_normal()*, *estado_caer()* y *estado_mover()*. En cada una de estas funciones habría que implementar el comportamiento deseado de nuestro actor para ese estado concreto. En el siguiente código se muestra un ejemplo del método *estado_mover()*, que se encargaría de desplazar horizontalmente al actor:

```cpp
void estado_mover(void) {
    mover(_x + _vx, _y);
}
```

Implementando de esta manera un método por cada estado, únicamente habría que definir el método virtual puro *actualizar()* para que, según el estado actual del actor, se ejecute la función correspondiente.

Lo ideal es organizar el comportamiento del actor en un autómata finito determinado, donde se especifiquen los estados posibles, y las transiciones que pueden darse entre los distintos estados. Hay que mencionar que los cambios de estado se realizarán desde una clase derivada de *Nivel*, que será el escenario donde los actores se encontrarán. El motivo de esta decisión no es otro que la falta de conocimiento que tiene un actor sobre lo que ocurre a su alrededor en el escenario del juego, información que sí está disponible en todo momento en la clase que se encarga de gestionar éste.

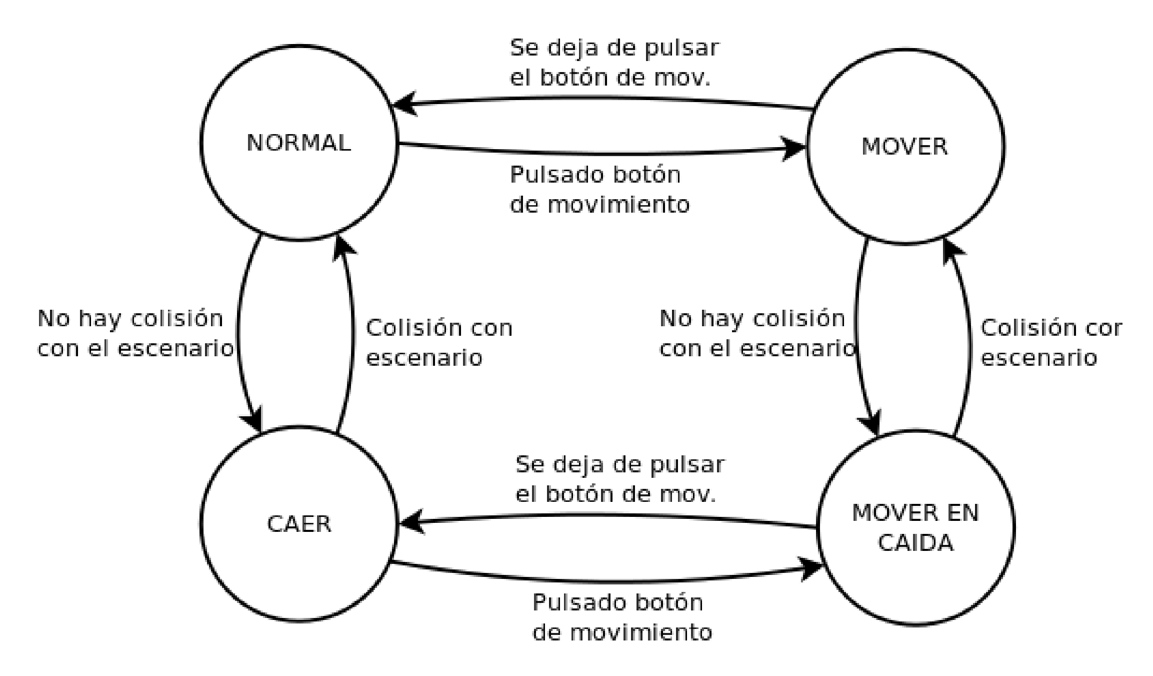


Figura 3: Sencillo autómata de ejemplo para el comportamiento de un actor

En la figura 3 puede apreciarse un sencillo ejemplo de autómata finito determinado, formado por cuatro estados, y que define el comportamiento de un actor controlado por un jugador a través de un mando. El actor comienza en el estado denominado *NORMAL*, en el que no sufre ninguna modificación de sus variables internas de posición. Desde este estado, y dependiendo de las condiciones que se cumplan, se puede pasar a los estados *MOVER* (si el jugador pulsa el botón de movimiento) o *CAER* (si no existe colisión entre el actor y el escenario). En el primero, nuestro actor modifica su posición horizontal en base a su velocidad en este eje, y en el segundo, se cambia la posición vertical hacia abajo, en base también a la velocidad del actor respecto al eje vertical. Desde estos dos estados puede llegarse a *MOVER EN CAIDA*, que es una combinación de ambos (movimiento en ambos ejes).

Puede comprobarse que, en cada estado, se proporciona un comportamiento para el actor y una serie de condiciones para que sucedan cambios de estado del actor. La manera de comportarse del actor para cada estado debe programarse en el método *actualizar()* de la clase derivada de *Actor* que controla a éste; sin embargo, las comprobaciones sobre el cumplimiento de condiciones que deben satisfacerse para ejecutar un cambio desde un estado a otro puede realizarse en el método correspondiente de la clase que controle el escenario (*actualizarPj()* para los actores jugadores, o *actualizarNpj()* en el caso de los actores controlados por la máquina), o bien en el propio método de actualización del actor, ya que éste tiene un atributo que es una referencia al nivel en el que participa. Se recomienda optar por la

primera opción, para así separar el comportamiento del actor según el estado y las transiciones posibles entre éstos.

4.3. Niveles

Un nivel representa el escenario donde los actores, ya estén controlados por un jugador o por la máquina, interactúan entre sí y con los componentes de dicho nivel. Al igual que ocurre con los actores, *LibWiiEsp* proporciona una clase base para crear niveles de una manera sencilla y rápida, que permite diseñar cada nuevo escenario utilizando el software *Tiled*, editor de mapas de tiles.

Hay que distinguir entre los conceptos de escenario y nivel. Para *LibWiiEsp*, un nivel es una clase derivada de la clase abstracta *Nivel*, y que define varios escenarios que se comportan de la misma forma, siendo cada escenario un mapa de tiles generado con *Tiled* en el que se especifica la disposición de los tiles y los actores que participan.

Con esto se consiguen varias ventajas. En primer lugar, definiendo una sola vez el comportamiento de un tipo de escenario en una clase derivada de *Nivel*, se pueden generar múltiples escenarios cuya lógica sea la implementada en esta clase. Por otro lado, este sistema permite que, en un mismo videojuego, se puedan intercalar fácilmente escenarios con comportamientos distintos (como ejemplo, se puede pensar en las típicas fases de *bonus* de clásicos como *Street Fighter*, en las que se debe destruir un coche o varios barriles en lugar de luchar contra otro oponente controlado por la máquina).

4.3.1. Partes de un nivel

Un nivel se compone de tres partes, que son los distintos tipos de objetos que se cargan en un escenario. En primer lugar, se encuentran las *propiedades* del nivel, que son una serie de cadenas de caracteres que indican códigos de recursos en la galeria de medias del sistema; las propiedades del nivel son la imagen de fondo (que es fija, y permite dibujar un paisaje estático en la última capa de dibujo), la pista de música asociada a un nivel y la imagen del *tileset*. Pueden añadirse más propiedades según el videojuego que estemos desarrollando, pero la lectura y carga de esta información deberá ser programada en el constructor de la clase derivada.

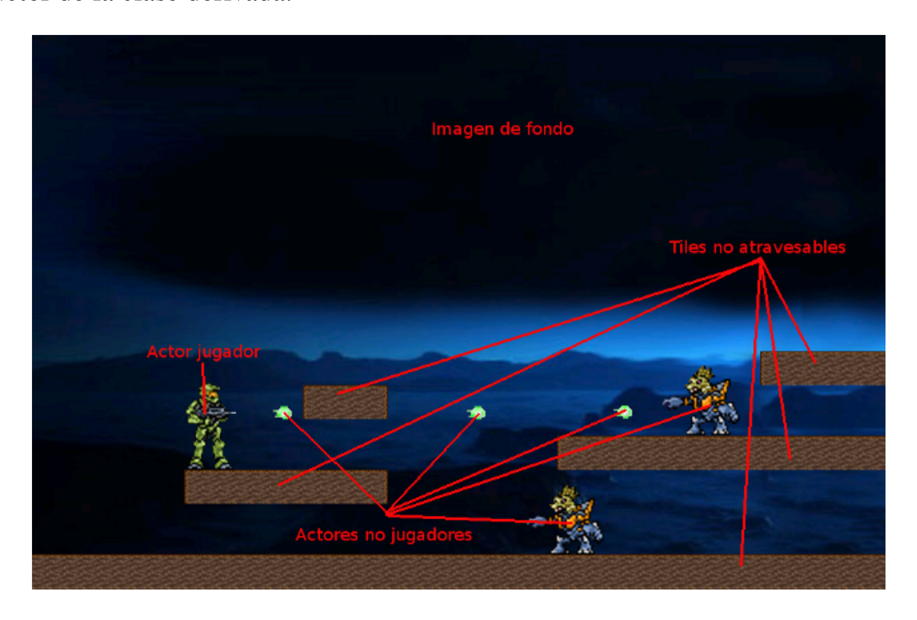


Figura 4: Distintas partes de un escenario

Llegados a este punto, es necesario explicar brevemente los conceptos de *tile* y *tileset*. Un tile es una imagen pequeña, generalmente cuadrada o rectangular, que se utiliza para componer un escenario en un videojuego en dos dimensiones. En un mismo escenario, se emplean numerosos tiles que se recogen en una única imagen organizados como una tabla (es decir, en filas y columnas). Esta imagen que almacena todos los tiles utilizados en un mismo escenario es lo que conocemos por tileset.



Figura 5: Ejemplo de tileset, formado por 6 tiles de 32x32 pixeles

Continuando con los tipos de objeto que componen un escenario, en segundo lugar tenemos la propia composición de éste, que está formada por dos capas de dibujo en las que se define la composición del escenario en sí a partir de los tiles del tileset asociado al nivel. Internamente, cada capa de dibujo se divide en una rejilla de cuadros (filas y columnas), donde cada celda tiene las mismas dimensiones que un tile, y en ella se coloca un tile concreto. Una de las dos capas del escenario, la llamada *PLATAFORMAS*, se caracteriza en que, cuando *LibWiiEsp* la carga en memoria, asigna a cada tile una figura de colisión de su mismo tamaño, de tal manera que los tiles que componen esta capa de dibujo pueden interactuar con los actores del juego. Por otro lado, la capa denominada *ESCENARIO* está compuesta por tiles que no tienen asociada ninguna figura de colisión, y únicamente se añaden al nivel con el objetivo de mejorar visualmente el escenario.

Generalmente, esta distinción entre tiles con figura de colisión asociada y sin ella se utiliza para definir los tiles que pueden ser atravesados por los actores del juego (aquellos que no tienen asociada una figura de colisión), y los que no permiten que los actores los atraviesen, que son los que sí tienen figura de colisión asociada.

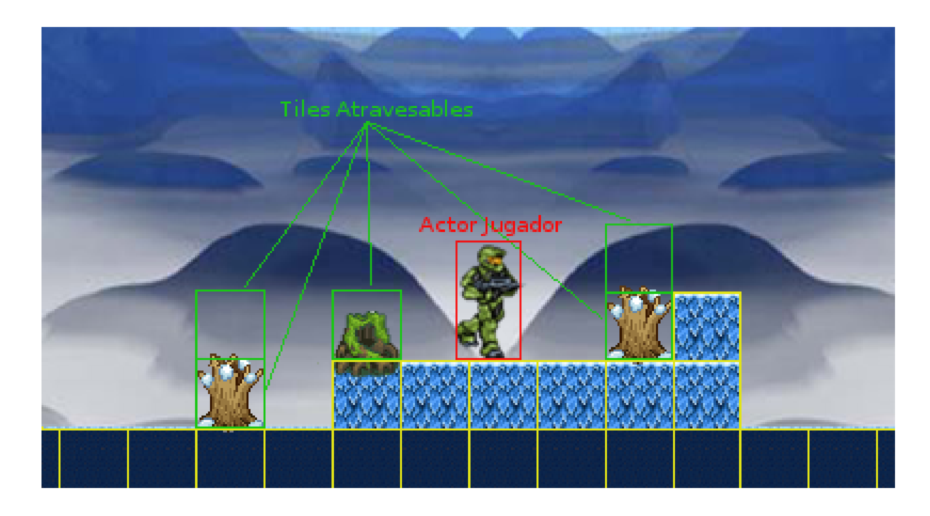


Figura 6: Ejemplo de escenario con tiles atravesables y no atravesables

Por último, nos encontramos con el último tipo de objetos que necesitamos para definir un escenario, que son los actores que participan en el nivel. Se distinguen los actores controlados por los jugadores, y

los que son dirigidos por la máquina.

Como ya se ha comentado en la sección de los actores, éstos tendrán definido su comportamiento según el estado en el que se encuentren en un momento determinado. Sin embargo, las transiciones entre distintos estados es conveniente realizarlas en el correspondiente método de actualización de la clase que gestiona el escenario, a pesar de que se pueden hacer desde la clase del actor.

4.3.2. Creación de un escenario con *Tiled*

El proceso de creación de un escenario a partir de la herramienta *Tiled* es muy sencillo. En su página oficial, *http://www.mapeditor.org/*, podemos descargar la última versión o, dependiendo de si el sistema en el que nos encontremos (en mi caso al redactar este manual, una Ubuntu 10.10) dispone de esta utilidad en los repositorios, se puede instalar con la siguiente orden:

```
sudo apt-get install tiled
```

Una vez instalado, ejecutamos *Tiled* y pulsamos en el botón *Nuevo*. Se abrirá una ventana en la que se nos preguntan algunos parámetros del nuevo mapa de tiles. Seleccionamos proyección ortogonal, el ancho y alto en píxeles de cada tile, y el ancho y alto medido en número de tiles que tendrá el escenario. Las medidas de un tile deben ser múltiplos de 8 píxeles, debido a los requisitos para la carga de texturas con *LibWiiEsp*, siendo un tamaño recomendable es 32x32 píxeles por tile. La resolución que utiliza la Nintendo Wii en un sistema PAL y proporción 4:3 es de 640 píxeles de ancho por 528 de alto, por lo que el tamaño mínimo del escenario debería ser (si utilizamos la medida 32x32 píxeles por tile) de 20 tiles de ancho, por 16 tiles de alto. Una vez establecidos estos parámetros, pulsamos en aceptar.

A continuación, hay que cargar en la herramienta la imagen que contiene el tileset. Para ello, pulsamos en *Mapa*, y después en *Nuevo conjunto de patrones*. Se abrirá un diálogo en el que debemos introducir el nombre que le daremos a la imagen (esto es un dato interno de *Tiled*, y el valor introducido aquí no es relevante), la propia imagen, y las medidas que tendrá un tile (ancho y alto en píxeles). Un detalle a tener en cuenta, en nuestra imagen no debe haber separación entre los tiles, ni tampoco margen. Al introducir estos datos, hacemos clic en *Aceptar* y ya tendremos el tileset listo para dibujar el mapa.

Como siguiente punto, hay que introducir las tres propiedades del escenario. Nos vamos a *Mapa* y entonces a *Propiedades del Mapa*, e introducimos en la lista estas propiedades:

- imagen_fondo: Su valor es el código que tendrá la imagen de fondo del escenario en la galería de recursos del sistema.
- imagen_tileset: Su valor es el código que tendrá la imagen del tileset del escenario en la galería de recursos del sistema.
- musica: Su valor es el código que tendrá la pista de música del escenario en la galería de recursos del sistema.

Tras introducir las propiedades, pulsamos en *Aceptar*, y procedemos a preparar las tres capas de dibujo de nuestro escenario, que deben llamarse obligatoriamente como se describe en la lista:

- escenario: Capa de patrones donde dibujaremos los tiles atrabvesables.
- plataformas: Capa de patrones donde dibujaremos los tiles no atrabvesables, es decir, los que tendrán asociada una figura de colisión.

- actores: Capa de objetos donde situaremos todos los actores que participen en el nivel.

En este punto, ya estamos en disposición de comenzar a dibujar nuestro escenario. Es recomendable comenzar por la capa de tiles no atravesables, continuar decorando con los tiles atravesables, y por último, establecer la posición de los actores. Para situar un actor en el escenario, insertamos un objeto en el lugar que deseemos, y lo redimensionamos adecuadamente para que se vea correctamente cómo quedaría su posición inicial. Seguidamente, hacemos clic con el botón derecho en el actor, y pulsamos en *Propiedades del objeto*. Se mostrará un diálogo en el que debemos indicar, en el campo *Tipo*, el tipo del actor que comenzará aquí (dato necesario a la hora de distinguir qué clase derivada de *Actor* hay que instanciar), y además la propiedad *xml*, cuyo valor debe ser la ruta absoluta hasta el archivo XML que contiene la información de este tipo de actor en la tarjeta SD de la Nintendo Wii. Además, si el actor es controlado por un jugador, debemos añadir también la propiedad *jugador* con el código identificativo del jugador que jugará con este actor como valor.

A la hora de definir los distintos actores, se puede utilizar el campo *Nombre* del diálogo de propiedades de un actor para identificarlo en tiempo de diseño del escenario.

4.3.3. Implementando una clase que controle escenarios

Una vez tenemos creados uno o varios escenarios cuyo comportamiento (la gestión de transiciones entre los estados de los distintos actores que participan en él, el movimiento o no del *scroll* de la pantalla sobre el escenario, etc.) es común, el último paso para poder disfrutar de ellos es definir una clase derivada de la abstracta *Nivel* que controle todos los detalles del nivel. Cada clase derivada definirá una gestión distinta de un grupo de varios escenarios.

El primer paso para crear una clase derivada de la abstracta *Nivel* es implementar el método virtual *cargarActores()*. Como se detalla en la documentación de la clase, en el constructor se cargan todos los tiles del escenario, se toma la imagen de fondo y del tileset desde la galería, y se leen los datos de inicialización de cada actor participante, almacenándose en una estructura temporal. La definición de este método debe recorrer esta estructura temporal de datos de actores, creando, para cada ocurrencia, un actor de la clase derivada de *Actor* correspondiente, tal y como se aprecia en el siguiente ejemplo:

```
void cargarActores(void) {
  for(Temporal::iterator i = _temporal.begin(); i != _temporal.end(); ++i)
  {
    if(i->tipo_actor == "jugador") {
      Personaje* p = new Personaje(i->xml, this);
      p->mover(i->x, i->y);
      _jugadores.insert(std::make_pair(i->jugador, p));
    } else if(i->tipo_actor == "bicho") {
      Bicho* p = new Bicho(i->xml, this);
      p->mover(i->x, i->y);
      _actores.push_back(p);
    }
  }
  _temporal.clear();
};
```

En el ejemplo, el programador ha definido dos clases derivadas de *Actor*, denominadas *Personaje* y *Bicho*. En el método implementado, se recorre la estructura temporal de datos de actores del escenario, y

se crea un actor a partir de la clase correspondiente (según el tipo de actor que se haya indicado desde *Tiled*), se mueve el actor hasta su posición en el escenario, y por último se inserta en la estructura adecuada (*_jugadores* en el caso de los actores controlados por un jugador, en la que hay que indicar también el código identificador del jugador concreto; o *_actores* para los actores controlados por la máquina).

Es importante recordar que este método *cargarActores()* se debe llamar desde el constructor de la clase derivada de *Nivel* con la intención de que la creación de los actores se realice en el momento de cargar el escenario. Además, es recomendable vaciar la estructura temporal cuando se finalice el proceso.

El siguiente paso en la generación de esta clase derivada es implementar los métodos de actualización del nivel, que se deberían llamar en cada fotograma del programa. A continuación se indican cuáles son, y qué funcionalidad se espera que tengan.

En el método *actualizarPj()* se debe actualizar el estado de un único actor jugador, atendiendo tanto al mando concreto que tenga asociado en la estructura *_jugadores*, como a la situación del escenario. Un ejemplo sencillo podría ser:

```
void actualizarPj(const std::string& jugador, const Mando& m) {

  // Estado NORMAL: puede pasar a MOVER
  if(_jugadores[jugador]->estado() == "normal") {
    if(m.pressed(Mando::BOTON_ARRIBA) or m.pressed(Mando::BOTON_ABAJO))
      _jugadores[jugador]->setEstado("mover");
  }

  // Estado MOVER: puede pasar a NORMAL
  if(_jugadores[jugador]->estado() == "mover") {
    if(m.pressed(Mando::BOTON_ARRIBA)) {
      _jugadores[jugador]->invertirDibujo(true);
      s16 vel_x = _jugadores[jugador]->velX();
      if(vel_x > 0)
        vel_x *= -1;
      _jugadores[jugador]->setVelX(vel_x);
    } else if(m.pressed(Mando::BOTON_ABAJO)) {
      _jugadores[jugador]->invertirDibujo(false);
      s16 vel_x = _jugadores[jugador]->velX();
      if(vel_x < 0)
        vel_x *= -1;
      _jugadores[jugador]->setVelX(vel_x);
    } else
      _jugadores[jugador]->setEstado("normal");
  }

  // Actualizar el actor en base a su nuevo estado actual
  _jugadores[jugador]->actualizar();
}
```

Por otro lado, el método *actualizarNpj()* debe recorrer la estructura en la que se almacenan los actores controlados por la máquina y actualizar los que se consideren oportunos (aquí se deja en manos del programador el actualizar todos los actores, sólo los que están en pantalla, o los que cumplan un determinado criterio). Como ejemplo, se muestra la siguiente función que actualizaría el estado de todos los actores no jugadores:

```
void actualizarNpj(void) {
  for(Actores::iterator i = _actores.begin() ; i != _actores.end() ; ++i) {
    // Estado NORMAL: puede pasar a CAER
    if((*i)->estado() == "normal")
      if(not colision((*i)))
        (*i)->setEstado("caer");

    // Estado CAER: puede pasar a NORMAL
    if((*i)->estado() == "caer")
      if(colisionVertical((*i)))
        (*i)->setEstado("normal");

    // Actualizacion del actor
    (*i)->actualizar();
  }
};
```

El último método a implementar es *actualizarEscenario()*, en el que se espera que se implementen todos los demás detalles relativos al escenario que necesiten ser actualizados a cada fotograma del juego. El siguiente ejemplo muestra una implementación que únicamente actualiza el *scroll* de la pantalla sobre el escenario, según la posición horizontal del jugador cuyo código identificador es *pj1*:

```
void actualizarEscenario(void) {

  if(_jugadores["pj1"]->x() - _scroll_x >= screen->ancho() / 2)
    moverScroll(_scroll_x + abs(_jugadores["pj1"]->velX()), _scroll_y);
  else if(_jugadores["pj1"]->x() - _scroll_x <= screen->ancho() / 4)
    moverScroll(_scroll_x - abs(_jugadores["pj1"]->velX()), _scroll_y);

  if(_jugadores["pj1"]->y() - _scroll_y >= screen->alto() / 2)
    moverScroll(_scroll_x, _scroll_y + abs(_jugadores["pj1"]->velY()));
  else if(_jugadores["pj1"]->y() - _scroll_y <= screen->alto() / 4)
    moverScroll(_scroll_x, _scroll_y - abs(_jugadores["pj1"]->velY()));
};
```

Por supuesto, quiero remarcar que los métodos de ejemplo son precisamente eso, ejemplos muy sencillos cuya finalidad es que sirvan de guía para comprender cómo se trabaja con la plantilla de niveles de *LibWiiEsp*, y a partir de los cuales poder desarrollar los métodos de actualización necesarios (al derivar la clase abstracta *Nivel* se pueden añadir los métodos que se consideren necesarios).

4.4. Juego

La clase abstracta *Juego* es la tercera y última plantilla que *LibWiiEsp* ofrece para facilitar el desarrollo de videojuegos para Nintendo Wii. Es muy sencilla, y consiste en dos partes principales. El constructor se encarga de inicializar la consola a partir de la información que se introduzca en el archivo de configuración de la aplicación, y el método *run()* ejecuta el bucle principal del programa. A continuación se aportan todos los detalles relativos a esta plantilla para construir la clase principal de nuestro videojuego.

4.4.1. Inicialización de la consola

Como ya se ha comentado, la inicialización de todos los sistemas de la consola Nintendo Wii se realiza en el constructor de la clase *Juego*, que recibe como parámetro la ruta absoluta en la tarjeta SD de un archivo XML de configuración. Esta inicialización consiste en montar la primera partición de la tarjeta SD de la consola (debe tener un sistema de ficheros FAT), establecer el sistema de *logging*, leer el archivo de configuración y, a partir de éste, iniciar todos los aspectos de la consola que vamos a utilizar.

El proceso de inicialización, llegados a este punto, es el siguiente:

1. Inicializar la pantalla, el sistema de mandos, el sonido y las fuentes de textos (en este orden).

2. Establecer el color transparente, y los fotogramas por segundo que tendrá la aplicación.

3. Cargar los identificadores de los jugadores y asociar un mando con cada uno de ellos.

4. Cargar en memoria todos los recursos multimedia que se indiquen en el archivo XML de la galería.

5. Cargar en memoria las etiquetas de texto del soporte de idiomas.

Cuando creamos una clase derivada de *Juego* hay que llamar al constructor de la clase base, pasándole como parámetro la ruta absoluta en la tarjeta SD del archivo de configuración. Por otro lado, el destructor de la clase base se encarga de liberar la memoria ocupada por la estructura que almacena los objetos de la clase *Mando*, y de llamar a la función *exit()*, hecho obligatorio para que la pila de la función *atexit()* se ejecute al salir del programa (esto es muy importante, ya que en caso contrario nos encontraremos con una pantalla de error por no haber apagado los sistemas de la consola).

Un ejemplo del archivo XML de configuración es el siguiente:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<conf>
  <log valor="/apps/wiipang/info.log" nivel="3" />
  <alpha valor="0xFF00FFFF" />
  <fps valor="25" />
  <galeria valor="/apps/wiipang/xml/galeria.xml" />
  <lang valor="/apps/wiipang/xml/lang.xml" defecto="english" />
  <jugadores pj1="pj1" pj2="pj2" pj3="" pj4="" />
</conf>
```

En este archivo de configuración se establece que el sistema de *logging* registrará todos los eventos que sucedan en el sistema, el color transparente será el magenta, se correrá la aplicación a 25 fotogramas por segundo, se indican los archivos XML de la galería y el sistema de idiomas, se establece el inglés como idioma por defecto, y se prepara la consola para trabajar con dos mandos, asociados respectivamente a un jugador identificado por el código *pj1* y otro identificado por *pj2*.

4.4.2. El bucle principal

La clase base *Juego* proporciona, además, un método que ejecuta un bucle principal sencillo. Este método es virtual, de tal manera que si el programador necesita otra forma de gestionar su aplicación, se le permite redefinirlo en su clase derivada.

El método *run()* es el que se encarga de controlar este bucle principal. En primer lugar, llama al método *cargar()*, que es virtual puro y debe ser definido en la clase derivada de *Juego*. En esta función debe ejecutarse todo lo que se necesite **antes** de que se entre en el bucle. Después, se inicializa la bandera de salida con un valor falso y se establece el contador de ciclos del procesador a cero (este contador se utiliza para mantener constante el valor de los fotogramas por segundo), tras lo cual se entra en el bucle principal.

El bucle principal actualiza, al principio de cada fotograma, el estado de todos los mandos conectados a la consola, ejecuta el método virtual puro *frame()*, y después finaliza el fotograma y controla la tasa de FPS. En este método *frame()* se incluirán todos los detalles de la ejecución de cada fotograma, y devolverá un valor booleano falso si la ejecución debe continuar, siendo el valor de retorno verdadero en el caso de que el programa deba terminar.

Un detalle más, en el caso de que ocurriera una excepción en el transcurso de la ejecución del programa, ésta será registrada por el sistema de *logging* (siempre que éste esté activado, al menos, en el nivel *error*, identificador 1), y después se saldrá de la aplicación.

Por último, destacar el hecho de que tanto si se necesita una gestión del bucle principal diferente, o un mayor número de funciones, el hecho de tener que derivar de la clase *Juego* implica la posibilidad de crear tantos métodos como sea necesario, y la redefinición opcional del método *run()* nos permite ejecutar estos nuevos métodos de la manera que mejor se adecúe a nuestras necesidades.

5. GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying

with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.